Review from Lecture 15
- Standard library sets
- Steps in program design
- Conway’s Game of Life

Today’s Lecture — Pointers and Arrays
Koenig and Moo, Section 10.1; Malik, pages 742-756
- Pointers
- Arrays, array initialization and string literals
- Arrays and pointers

16.1 Overview
- Pointers store memory addresses
- Pointers are the iterators for arrays
- Pointers are also the primitive mechanism underlying vector iterators, list iterators, and map iterators.
- Dynamic memory is accessed through pointers.

16.2 Pointer Example
- Consider the following code segment:

```cpp
float x = 15.5;
float *p;
p = &x;
*p = 72;
if (x > 20)
    cout << "Bigger\n";
else
    cout << "Smaller\n";
```

The output is **Bigger**
because **x == 72.0**. What’s going on?

16.3 Pointer Variables and Memory Access
- `x` is an ordinary integer, but `p` is a pointer that can hold the memory address of an integer variable. The difference is explained in the picture above.
- Every variable is attached to a location in memory. This is where the value of that variable is stored. Hence, we draw a picture with the variable name next to a box that represents the memory location.
- Each memory location also has an address, which is itself just an index into the giant array that is the computer memory.
- The value stored in a pointer variable is an address in memory. In this case, the statement:

```cpp
p = &x;
```

Takes the address of `x`'s memory location and stores it (the address) in the memory location associated with `p`. 
• Since the value of this address is much less important than the fact that the address is \( x \)'s memory location, we depict the address with an arrow.

• The statement:

\[
*p = 72;
\]

causes the computer to get the memory location stored at \( p \), then go to that memory location, and store 72 there. This writes the 72 in \( x \)'s location.

• The distinction between \( p \) and \( *p \) for pointers is just like the distinction between \( p \) and \( *p \) for iterators.

16.4 Defining Pointer Variables

• In the example below, \( p \), \( s \) and \( t \) are all pointer variables (pointers, for short), but \( q \) is NOT. You need the \( * \) before each variable name.

\[
\text{int } * \ p, \ q; \\
\text{float } *s, \ *t;
\]

• There is no initialization of pointer variables in this two-line sequence, so a statement below will cause some form of “memory exception”. This means your program will crash!

\[
*p = 15;
\]

16.5 Operations on Pointers

• The unary operator \( * \) in the expression \( *p \) is the “dereferencing operator”. It means “follow the pointer”

• The unary operator \& in the expression \( &x \) means “take the memory address of.”

• Pointers can be assigned. This just copies memory addresses as though they were values (which they are). Let’s work through the example below. What are the values of \( x \) and \( y \) at the end?

\[
\text{float } x=5, \ y=9; \\
\text{float } *p = &x, \ *q = &y; \\
*p = 17.0; \\
*q = *p; \\
q = p; \\
*q = 13.0;
\]

• Assignments of integers or floats to pointers and assignments mixing pointers of different types are illegal. Continuing with the above example:

\[
\text{int } *r; \\
r = q; \quad // \text{Illegal: different pointer types;} \\
p = 35.1; \quad // \text{Illegal: float assigned to a pointer}
\]

• Comparisons between pointers of the form \( \text{if } ( \ p == q ) \) or \( \text{if } ( \ p != q ) \) are legal and very useful! Less than and greater than comparisons are also allowed. These are useful only when the pointers are to locations within an array.

16.6 Exercise

• What is the output of the following code sequence?

\[
\text{int } x = 10, \ y = 15; \\
\text{int } *a = &x; \\
\text{cout } \ll \text{ } x \ll \text{ " } \ll y \ll \text{ endl;} \\
\text{int } *b = &y; \\
\text{a } = \text{x } \ast \text{b;} \\
\text{cout } \ll \text{ } x \ll \text{ " } \ll y \ll \text{ endl;} \\
\text{int } *c = b; \\
\text{c } = 25; \\
\text{cout } \ll \text{ } x \ll \text{ " } \ll y \ll \text{ endl;} 
\]
16.7 Null Pointers

- Pointers that don’t (yet) point anywhere useful should be given the value 0, a legal pointer value.
  - Most compilers define NULL to be a special pointer equal to 0.
- Comparing a pointer to 0 is very useful. It can be used to indicate whether or not a pointer has a legal address.
  (But don’t make the mistake of assuming pointers are automatically initialized to 0.) For example,
  
  ```
  if ( p != 0 )
  cout << *p << endl.
  ```

  tests to see if p is pointing somewhere that appears to be useful before accessing the value stored at that location.
- Dereferencing a null pointer leads to memory exceptions (program crashes).

16.8 Arrays

- Here’s a quick example to remind you about how to use an array:
  ```
  const int n = 10;
  double a[n];
  int i;
  for ( i=0; i<n; ++i )
    a[i] = sqrt( double(i) );
  ```

- Remember: the size of array a is fixed at compile time. vectors act like arrays, but they can grow and shrink dynamically in response to the demands of the application.

16.9 Pointers As Array Iterators

- Pointers are the iterators for arrays.
- The array initialization code above, can be rewritten as:
  ```
  const int n = 10;
  double a[n];
  double *p;
  for ( p=a; p<a+n; ++p )
    *p = sqrt( p-a );
  ```

  Does this look vaguely familiar? It is similar to what you have already seen with vectors and vector iterators.
- But, the assignment:
  ```
  p = a;
  ```

  is different. It takes the address of the start of the array and assigns it to p. This illustrates the important fact that the name of an array is in fact a pointer to the start of a block of memory. We will come back to this several times! We could also write this line as:
  ```
  p = &a[0];
  ```
- The test p<a+n checks to see if the value of the pointer (the address) is less than n array locations beyond the start of the array. We could also have used the test p != a+n
- By incrementing, ++p, we make p point to the next location in the array.
- In the assignment:
  ```
  *p = sqrt( p-a )
  ```

  p-a is the number of array locations between p and the start. This is an integer. The square root of this value is assigned to *p.
- We will draw a picture of this in class.
- Do you see how pointers are just like vector iterators?
16.10  Exercises

For each of the following problems, you may only use pointers (array iterators) and not subscripting:

1. Write code to print the array \( a \) backwards, using pointers.

2. Write code to print every other value of the array \( a \), again using pointers.

3. Write a templated function that checks whether the contents of an array are sorted into increasing order. The function must accept two arguments: a pointer (to the start of the array), and an integer indicating the size of the array.

16.11  Character Arrays and String Literals

• In the line
  ```
  cout << "Hello!" << endl;
  ```

"Hello!" is a string literal. It is also an array of characters (with no associate variable name).

• A char array can be initialized as:
  ```
  char h[] = {'H', 'e', 'l', 'l', 'o', '!', '\0'};
  ```
  or as:
  ```
  char h[] = "Hello!";
  ```

In either case, array \( h \) has 7 characters, the last one being the null character.

• The C and C++ languages have many functions for manipulating these “C-style strings”. We don’t study them much anymore because the standard string library is much more logical and easier to use.

• One place we use them is in file names and command-line arguments.

16.12  Conversion Between Standard Strings and C-Style String Literals

• We have been creating standard strings from C-style strings all semester. Here are 2 different examples:
  ```
  string s1( "Hello!" );
  string s2( h );
  ```

where \( h \) is as defined above.

• You can obtain the C-style string from a standard string using the member function \texttt{c\_str}, as in \texttt{s\_c\_str()}.